NOTE – this 2017 paper accurately reflects motivations and long-term goals.
It is outdated regarding implementation details and language grammar.
See lexon.org.

LEXON

LEGAL SMART CONTRACTS

Henning Diedrich[1]

9 September 2017

Abstract

A language and *virtual machine*[2] is proposed to allow lawyers, lawmakers and judges to read and write legally correct *smart contracts*[3] that can build on comprehensive legal code libraries and can express and adjust to court decisions, fictions, changes in regulations, arbitration and the differences between jurisdictions. We make the point that a different language is needed for smart contracts and why going forward, reversibility will be provided by the system layer, while retaining irreversibility on its lower, foundational parts. The proposed language structure, grammar and vocabulary are showcased and its compilation[4] rules are listed. We lay out its multi-lingual and -jurisdictional potential and the mechanics of the reversibility layer. We close with an outlook on the potential and notes on required governance and involvement of governments. The appendix has a brief survey about what some relevant smart contract languages look like and what their focus is. The intended audience are the legal profession and programmers. A best effort is made to support both sides.

1 INTRODUCTION

It should be possible to write[5] smart contracts like this:

```
ARTICLE: WITHDRAWAL.
Comment: This article describes how collected funds can be withdrawn.

If Requester is Beneficiary,
and Sum of Funds greater or equal Funding Goal,
and Funding Deadline past,
then: transfer Amount Raised to Beneficiary.

If this transfer succeeds,
then: record 'FUND TRANSFER', Beneficiary, Amount Raised;
else: allow Withdrawal By Funders.
```

The blockchain virtual machine should be able to 'understand' and execute this code.

[1] With contributions from Thomas Hardjono, Carla Reyes, Florian Glatz, Oliver Goodenough, Harald Stieber, James Hazard and Iason Vasileiou.

This example is a part of a crowdfunding[6] contract. The contract on the whole allows a *Beneficiary* to set a *Funding Goal* and a *Funding Deadline*. When after the deadline the funding goal is met, by payments made by *Funders*, the beneficiary will collect the money. This example shows the definition of the process of that withdrawal by the beneficiary. The *Requester* is the party that initiated the request that the payout be made to them. The code checks that the requester is indeed the *Beneficiary* that was set up at the outset to be the only identity eligible to receive the payout.

However, the funds could be distributed any *other* way seen fit – not at all described in the code above – by a judge or an arbitrator at any time and the contract can be made to adapt to these changes during a pre-set *arbitration window*, e.g. 30 days, using the *arbitration key* for this contract. This could be done to bring the smart contract state[7] and behavior[8] back in sync with the law, but also to debug its logic or setup. Funds are implicitly frozen until the arbitration window has passed. The window can be set to zero, resulting in smart contracts as we are used to today. Accordingly, the above 'transfer' goes into an implicit escrow to prevent ripple effects as a side effect of arbitration.

Arbitrators need not be made aware of their power to change the smart contract until needed and can essentially *change the code* of the contract *retro-actively* during the arbitration phase to solve any problems that arose. The arbitration key is a *public key*[9] associated with the contract that the arbitrators published. The same key can safely be re-used across an unlimited number of contracts. It can be changed by the arbitrators to make another party an arbitrator. It can be a key that requires multiple signatures of certain parties or the majority of a group of possible signers. There is no preparation needed on the side of the arbitrators if they have a public key published. We will likely see courts and arbitration tribunals publish their public keys for inclusion as arbitration keys, so as to be elected as effective arbitrators for a smart contract, analogous to writing a choice of law and choice of venue clause in a prose contract.

The above code would currently be written like this[10] *without any reversibility* or option for arbitration:

```
function withdrawal() {
      if (beneficiary == msg.sender
     && amountRaised >= fundingGoal
     && now >= deadline) {
          if (beneficiary.send(amountRaised)) {
           FundTransfer(beneficiary, amountRaised, false);
           } else {
                funderWithdrawalsAllowed = true;
           }
      }
}
```

[2] A **virtual machine** is a program that executes programs. This is how all blockchains work today: code to process a transaction or a smart contract is stored on the blockchain in a specific format, for example for the Ethereum Virtual Machine (EVM), and executed by the virtual machine to perform the transaction or execute the smart contract.

[3] A **smart contract** is a program that runs on a blockchain, cannot be stopped by the contractors and can make payments in cryptocurrency. For a non-technical introduction to blockchains and smart contracts, see Diedrich, *Ethereum*.

[4] A **compiler** translates high-level, human-readable *source code* into a machine readable format that is harder to interpret

While this is easy to read for programmers,[11] the point we are making is that this will never be read by judges without engaging a human translator; that laws or regulations will never be expressed this way without meticulous, costly and error-prone documentation overhead; and that this notation is unnecessarily technical for something aspiring to be a contract. *Unnecessary* meant in a literal way here.

The goal of this proposal is to empower lawyers,[12] arbitrators and judges, as members of the legal profession are needed to interpret smart contracts. To bring down costs and allow the new technology to scale, they need to be able to take part in the contract crafting in a more direct way.

Once code libraries emerge that encapsulate legal knowledge, those will be able to do some of the work that previously had been done by lawyers using word processors. The appreciation and governance of such libraries as a common good is a high goal. Eventually, regulations and laws could themselves be written in the proposed language.

for humans, the *opcodes*. The virtual machine executes the opcodes. Also see footnotes 46 and 47 on page 15.

[5] Only two lawyers have expressed their conviction that lawyers will *write* in such language in the future: Luka Müller-Studer of MME and Simon Polrot of Fieldfisher.

[6] **Crowdfunding** is a method where an entrepreneur collects micro-investments from *backers* promising in return memorabilia, early delivery and visibility into the founding and development process of the company and product.

[7] **State** is basically the contents of the variables and storage of a program at any given time, and is expected to change during execution, as opposed to its program code that it is usually not expected to change.

[8] **Behaviour** is the way a program acts out, governed by its code *and* the code in the libraries and frameworks it uses.

[9] A **public key** is one half of the ubiquitous cryptological means of *private/public key pairs*. Every secure web page transmission and every digital signature uses this technique. The public key can be used to verify signatures made with the secret private key. In blockchains, this is used to grant exclusive authorization to an action that only the holder of the private key will be able to perform then. To put the gate-keeping mechanism in place, only the public key is required.

[10] In the language *Solidity* that is used to program the prevalent blockchain of today, *Ethereum*.

[11] This "curly braces" style of notation is not idiosyncratic to Ethereum but emerging as universal standard for program code. It is based on the rather accessible *ECMAScript* that was developed in 1995 by Brendan Eich for the *Netscape* browser; inheriting its looks from the 1958 language *ALGOL*, as such being similar to the current biggest mainstream languages, *C++* and *Java* (see footnotes 21 and 23); standardized as ISO 16262:2011 – http://standards.iso.org/ittf/PubliclyAvailableStandards/c055755_ISO_IEC_16262_2011(E).zip.

[12] Houman Shadab, a professor at New York Law School, has pointed out that "By requiring parties to strictly commit, at the outset, to decisions of a smart contract, the need for transactional attorneys and others to structure smart contractual relationships may increase. Parties would most likely want to specify a more detailed range of contingencies and outcomes ahead of time before committing themselves to abide by the decisions of a software-driven contract." – quoted after Silverberg et al, 2016, *Getting Smart: Contracts on the Blockchain* – https://www.iif.com/file/15761/download?token=x1RUm_OF

INDEX

2 Rationale

Blockchain smart contracts, employing automation fortified by distributed computer systems, promise to significantly speed up and lower the cost of legal processing and arbitration in mundane contracts. This in turn lowers the cost of access to the legal process and its related functions, which will increase the health of our social fabric and boost the innovative and value creating elements in our economies. For this to happen, smart contracts need to be legally correct, which requires that they be comprehensible by members of the legal trade.

2.1 The Cost of the Legal System

Today, the high overhead cost of policing, and access to justice, make many otherwise viable niche markets impossible, both in the real and the virtual world, and skew playing fields to the benefit of big players. The unsustainable reality is that the threshold from which court action begins to make sense lies at six figures,[13] below which the economics of attorney and court fees[14] work in favor of frivolous abusers and outright frauds. The resulting loss in commercial activity, human suffering from experienced injustice and stifling of entrepreneurial capacity damages the economy and erodes an essential pillar of peace and trust in our societies to a staggering degree. This is recognized by the Access to Justice movement and this paper can be a contribution to the effort by providing an approach that is concrete, scales and works for small-time situations as well as taking on the customary ways of entrenched stakeholders in a big way, supported by the force of a technological quantum leap.

A natural limit exists in the processing capacity of the human brain. Cases beyond a threshold of complexity take a long time to communicate from client to lawyer, can be too deep to penetrate for a lawyer within a short time and in their details repeatedly be forgotten or remembered wrongly over the course of a trial. These hard human limits of brainpower of lawyers justify high price tags, explain egregious consequences of misunderstandings between lawyer and clients that deepen the divorce between justice and law. It is obvious how machines excel at helping with this type of complexity challenges and are doing so in many other industries.

An urgency to act exists within large corporations: while the factual side of relationships between businesses are primarily expressed through contracts, legal departments routinely have a lack of understanding of the fundamentals and the spirit of a collaboration and a detrimental effect on the development of, and the results from interactions between corporations. A preference for the predatory perspective of a zero-sum game[15] rather than a mutual beneficial symbiosis, often divorces the take of the lawyers at a company from those of the business people and workers who actually produce value and are interested in win-win situations – to create a sustainable business and as expression of the meaning inherent to their work. For decades, efforts have come up short to reign in the sprawl of copy-pasted legal texts and negativity of the contributions of legal experts, and to manage the chaos caused by millions of pages of contracts being produced.

[13] Pick your favorite fiat currency, USD, EUR or GBP.

[14] This independently of who *in theory* pays the lawyers. Where regulation exists, a disinterested lawyer working for mandatory, low fees is the worst. The good ones will always ask you to waive those limits for them to have more fun.

[15] James Hazard, *Common Accord* – http://www.commonaccord.org/

2.2 Time for Change

The mirror image of this travesty is the indignation with which not a few number of lawyers react to the mention of the notion of *justice*. They are trained to acquire this mindset and while their scorn is rational and helping with the necessity to acquire work across the spectrum of cases, it is at best cynical and at worst the expression of unfettered corruption that in its severe consequences for other people has a corrosive effect on the social fabric as well as the productivity of industries.

There is, despite what lawyers are taught and like to believe, perhaps a much better way.

Law as understood, practiced and taught within a given nation is an on-going social process that involves people across generations. The various legal substrates that have accumulated will require time to further evolve based on new social thinking and new technological innovations. Key to this is the education of the coming generation of legal practitioners. And core to this education is the broad understanding of the role of smart contracts and human-friendly expressions of smart contracts.

2.3 Improving Access to Justice

Blockchain technology can reduce the cost of access to justice and the legal system and even make policing unnecessary by its capacity to *enforce rules at very low cost*. For certain kinds of legal agreements, automation combined with immutability allows mundane tasks relating to the creation, execution and verification of legal contracts to be done at the costs of mere cents. In some circumstance, individuals may even engage in legally binding transactions without the mediation of human legal professionals.

Blockchain smart contracts that transfer crypto tokens are the ultimate enablers for fair subcontracting, bedrock trade finance, a new class of safe derivatives, yet unheard of niche markets, machine-to-machine commerce and much cheaper access to justice. Their power lies in their ability to *digitally transfer possession according to predefined rules*, with almost perfect reliability and at minimal cost. Crypto tokens combine bearer[16] qualities and safety with complete auditability. Where employed they immediately enable a secondary market.

For more complex legal situations in which a legal precedent or previous cases are rare or do not exist, smart contracts technology together with artificial intelligence may provide support for human legal experts to arrive more efficiently at a solution.

2.4 Primacy of Law

However, in most parts of the world,[17] smart contracts are not legally binding contracts at this point.[18] They are both more, and less and most of all, the unstoppable actions of a smart contract can easily be

[16] Bearer instruments are like cash: who holds them, owns them basically. Because that is so well suited for tax-evasion, they are banned in almost all jurisdictions. But bearer instruments have a massive advantage: to the bearer, there is perfect transparency what the exposure is. The loss can be at most exactly what the value of the asset is. There are no collaterals, no need for trust, no leverage: they are what is on the table. (Pindar Wong)

[17] Most jurisdictions are based on Civil Law, which as a rule allows for less contractual freedom than Case Law. Judges routinely strike out entire paragraphs of contracts that do not align with contract law. In economically skewed markets like flat leases, lease-seekers are encouraged to sign contracts that have unenforceable clauses because they are certain to be struck as if they had never been included, even if the signatory to the contract knew about it and signed anyway.

[18] Notwithstanding the legislation in Arizona, Delaware and Singapore where smart contracts are deemed legal agreements now. But see Werbach, Cornell, *Contracts Ex Machina*: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2936294

found unlawful. Their potential to revolutionize commerce will be hampered as long as they can be legally attacked for their nature, or for technical or legal mistakes in their programming. They need to be *comprehensible*[19] to lawyers and judges to ensure that they can be crafted according to the law, well understood by the entering parties and arbitrated according to their spirit.

Law trumps crypto in business and the legality of automated transfers can always be contested in court. Clarity and maximal human readability of smart contracts will reduce errors and frauds but also curb incentive for frivolous lawsuit and give corporations the best insurance against class actions. Device producers cannot afford to ship gadgets that enter into business relationships with each other, that could be found to be serially breaking the law. They need their lawyers to understand first hand how the devices are allowed to trade.

2.5 Reversibility

Since judges and lawmakers have the ability to retro-actively affect ownership, smart contracts need to have the ability to account for this reality, which is at odds with the irreversibility that they are currently based on. Uncontrolled ripple effects through the accounts of third parties have to be avoided though to maintain trust in the blockchain currency. But as with database transactions, reversibility can be incredibly laborious or even impossible to establish on the application layer – i.e. in the smart contract code itself – if it is not provided by the system layer underneath the smart contracs.

We are proposing a different type of blockchain system where reversibility of smart contracts can be achieved so that it reflects the decisions of a human judge and records them on the blockchain as post-event outcomes. The blockchain thus becomes a medium that captures *legal history* together with the *current* state of the legal process. Different layers of of this blockchain are proposed where a certain degree of reversibility can be effected at the higher-layer blockchain system, with increased degrees of permanence and irreversibility further down the lower-layer blockchain system.

2.6 Productivity

It is *productivity* and *error-prevention* that drives progress[20] in languages, programming environments, compilers and virtual machines. Given that Ethereum is Turing-complete, basically anything can be programmed in it. But this does not mean that everything can be programmed in a way that is still humanly understandable, debuggable, maintainable or performant enough to be usable. This is true for all recent program language paradigm shifts, e.g. from procedural to object oriented, or for virtual machine technology, e.g. garbage collection. These changes made new things possible because of the productivity they enabled, but in every single case the deprecated technology could *theoretically* have been used to implement anything that the new languages would excel at: there is nothing that can be programmed

[19] Oliver Goodenough notes that the presentation could, for example, also be visual, in form of flow diagrams. And in fact, the language proposed here will serve as ideal medium to empower visual and even point-and-click tools to create contracts, because it is closer to the business logic than the existing smart contract languages but unambiguously translatable into smart contract code that can be executed on the blockchain.

[20] A study by Motorola, comparing the languages *Erlang* and *C++* to find out whether Erlang was a more productive language concluded that Erlang was measurably more productive and reusable: “Can productivity and maintainability be improved? Yes, using source lines of code (SLOC) as a metric, both the Erlang DM and DCC are less than a third of the size of the C++ counterpart ... Moreover, much of the Erlang DCC is a *reusable* generic server ... The reasons for the reduced programming effort are that *coding for the successful case saves 27%*” (italics added).
https://www.researchgate.net/publication/221211369_Comparing_C_and_ ERLANG_for_motorola_telecoms_software

in C++[21] that could not have been programmed in C[22] and yet C++ replaced C as language and ecosystem of choice for many areas. Java[23] then did not allow for new things but was embraced for application programming where C++ had been used before. In the end, everything could be programmed in the most primitive of languages, machine code. It is no different with our proposal and therefore, it is not a sufficient argument that any smart contract should be implementable in Solidity and that the Ethereum virtual machine would be fully capable to serve as base layer for reversibility implemented on the application level, as part of the smart conracts' code.

Neither tolerable performance, let alone readability of smart contracts for lawyers and judges could be achieved this way. The resulting code would inevitably be hard to read and understand even for programmers. And while much functionality can be hidden in libraries, some semantics have to be implemented on the language and virtual machine level. As an example, it is possible to program loops in programs without loop structures.[24] Bitcoin, which has no loops, is probably as a system nevertheless Turing-complete. But the results are hideous and it is a last resort that is mostly unreadable and hard to make error free. The same is true for reversibility. It is no coincidence that on a lower level, Solidity has *exceptions*[25] that can roll an entire transaction back while it is still executing. A smart contract could be programmed that includes arbitration clauses for every single instruction of the 'happy path'.[26] But this would break readability completely and be exceedingly error-prone, as we are showing below.

2.7 Correctness

Error-freeness is important for any program but even more so for smart contracts that can lead to massive and irreversible loss of money when found lacking, as has been demonstrated by *The DAO*.[27] In the future they might be employed for tasks where more damage could result, e.g. by the release of information. Formal verification can only help to the degree that relevant questions about the verified code can be asked in math. Take, for example, the need to ensure that the two accounts in a transaction representing token ownership should, at its end, result into the same sum total as at the beginning of it – to verify that the algorithm of the transaction did not erroneously lose or produce tokens. On the syntactic level, the compiler itself can be supported by *linters*[28] that catch common typos and grammar errors before the program is executed. Correctness on the legal level however, cannot be achieved – at this point – without the understanding of a human, legal specialist. For this, the code has to be comprehensible.

[21] **C++** was developed as an extension of C, adding specific OO language elements to improve the productivity of programming teams and code re-usability, in effect to reduce cost and time to market but also to allow for more complex systems to be built in the first place.

[22] **C** is an old programming language from the 70ies that was made and is still used for programming operating systems and lower level tools like drivers. It predates OO and follows a *procedural* paradigm.

[23] **Java** is the current mainstream language for business programming. It allows to create programs with less attention to repetitive, low level memory management that is entirely separate from business logic.

[24] Programmer note: not talking about *tail recursion* here but e.g. *continuation passing style* in ECMA-like languages.

[25] See footnote 50.

[26] The part of the logical flow that is tread as long as all goes well, as opposed to the routes that exist for error handling and are generally hoped to be strode much less often. Which makes them relatively more expensive and less tested for errors.

[27] The DAO was a so-called decentralized, autonomous organization that was designed as an investment vehicle. It issued *tokens* and promised investors voting rights. It was brought down by an engineering mistake that was exploited by hackers who stole $56M. As a result, the Ethereum community reset part of the blockchain. See, *Ethereum, What Is the DAO?*

[28] **Linters** are tools that read software code, before it is compiled or run and flag syntax errors and suspicious constructs that are known to be error-prone. They typically find typos and also complex bugs that are very hard to spot for humans.

2.8 Language

As an example, object oriented programming[29] (OO) was introduced with the goal to make team work on large projects easier and code re-usable. One *could* program OO-style in non-OO languages.[30] It 'simply' takes following certain conventions and it has often been done, e.g. for environments where a C compiler had to be used because no C++ was available. There is no doubt that the essence of OO lies in how code and data are grouped together as objects, rather than what the language used provides to make that easier. In reverse, it often happens that less skilled developers program the older procedural style when using OO languages because they use them poorly. But this does not erase the strong advantage one gains when programming OO in OO languages that comes with their additional language constructs and also, from the *limitations* they intentionally introduce to enforce object orientation.[31]

From all this follows that the introduction of a new language and programming environment is the proven path to enable a new programming paradigm, for improved productivity, robustness and reusability.

2.9 Semantic Meaningfulness

A key goal of a lawyer-readable language is to facilitate the creation of meaningful legal agreements and contracts among parties. It provides a way to achieve early detection of contracts that are meaningless – in part or whole – and allow one party to question the intent of the counter party during the negotiation phase. But where unintelligible paper-based contracts will just have little or no legal enforceability, blockchain smart contracts that are non-sensical can be automatically detected through automated verification.

2.10 Joint Composition

Today in the paper-based world drafts of contracts are exchanged among transacting entities, and may undergo several rounds of revisions before all parties are satisfied. Languages such as the one proposed in this paper can facilitate more efficient contract creation through joint composition of contracts among parties. One party may suggest additions or changes to such contract, where the other party may then pass the contract through an automated verification tool that checks the semantic implication of these changes to find immediate answers to the question how the potential outcome in different cases has been affected by the proposed changes.

[29] The **Object oriented** programming (OO) paradigm postulates that code should be grouped together with data that it mostly operates on, and should be magically expandable without even looking at how it works internally. OO languages provide special constructs to enable this style.

[30] Sometimes called “Scandinavian OO.”

[31] As an example for limitations: OO languages often make it impossible to have global variables – variables that can be accessed from every piece of code throughout the entire program – so as to enforce the grouping of every bit of data to some specific code. This is called *syntactic salt a*s a play on *syntactic sugar* (see footnote 48).

3 Status Quo

3.1 Traditional Contracts as Process

There is an obvious correlation between (traditional) contracts and programs. Both deal in conditions, in inputs and results. When a judge tries to make sense of a written contract, she decomposes it into discrete, decidable chunks, just as a computer processor executes instructions step by step. The proof of the decomposability of contracts into automata was presented by Flood and Goodenough.[32]

This gave rise to the vision that contracts should be expressible as programs and the term *smart contract* was in fact first termed independently of blockchains, before they were a thing, to describe the possibility of a *more exact law*.[33] Lawyers are often quick to object and point out that contracts need wiggle room, that covering those last 2% of precision in the write up is insanely expensive and therefore usually left out because the customer was not interested in paying for that. It is also plausible that no one wants to get involved with insanely complex legal texts, because they are always insanely hard to read at any rate.

This objection is moot however, when the route of the exploration start from smart contracts, with the goal of bringing them closer to the legal sphere, as opposed to coming from the law at large and trying to find a way to formulate every possible law, regulation and contract as smart contracts. Not coincidently, research coming from that direction has by and large not arrived at the level of smart contracts yet. But when starting with smart contracts, we start with an inherently exact notation of something. The question then is, can the exactitude of the formulas that smart contracts are be expressed in legal terms?

They could, because law has the ambition to describe anything under the sun. That is the very reason why in reverse it seems so ambitious to try to develop the universal formula of law. This does of course not assume that all smart contracts, even those crafted in a thoroughly outlaw state of mind, would turn out to be *legal*. But they should always be *describable* in legal terms, sometimes in all their anarchic glory, in every detail that matters.

The bigger vision remains though, to codify laws and regulations of a jurisdiction so that contracts can tap into the existing source code produced by lawmakers. This is an incredibly wide terrain and to get there, we see no other option than to work in the close cover of very concrete use cases, such as blockchain smart contracts.

3.2 Blockchains

Smart contracts are enabled by blockchains and cryptocurrencies, which did not come about by chance. They were sought after since the late 80's as essential, technological corner stones to preserve personal freedom in the face of governments and corporations that started equipping themselves with computers. The decentralized timestamping feature of blockchains, for example, is not a newly discovered side effect of blockchains. The reverse is true. Timestamping was the very goal that hash–chaining was in-

[32] Flood, Goodenough, 2015, *Contract as Automaton: The Computational Representation of Financial Agreements* https://www.financialresearch.gov/working-papers/files/OFRwp-2015-04_Contract-as-Automaton-The-Computational-Representation-of-Financial-Agreements.pdf

[33] Nick Szabo, 1997, *Formalizing and Securing Relationships on Public Networks.* First Monday, Volume 2, Number 9 – http://ojphi.org/ojs/index.php/fm/article/view/548/469

vented for in the 90’s: proving that a document existed at a certain time without needing a centralized service for that. This was later used as a storage component in the creation of Bitcoin.

Many building blocks for blockchains and smart contracts were made by the *Cypherpunks*.[34] They also encouraged civil disobedience and engaged in activism. T-shirts were made with forbidden crypto[35] source code printed on them, programs printed as books to be able to legally export them. But most of all, *Open Source* programs were written that allowed for the free, private application of crypto, for example to encrypt emails. The full potential of these tools are slowly being understood by business and governments now.

3.2.1 Transaction Agency

The Cypherpunks defined the ability to make anonymous transactions – like using cash – as key to being able to have any privacy. It was as important to them as being able to encrypt communication. Without anonymous transactions, behavior patterns would still give away everything about you – what we now call *meta data* and what the agencies claim is harmless when snooped wholesale. The problem is that citizens are now being profiled more and more often based on arbitrary data patterns in which they fit. This is the opposite of equality.

Accordingly, the attempts to create a cryptocurrency were not a random strand of Cypherpunk activity, but from the start elemental to their cause. It was also clear to them that it doesn’t end with anonymous payment but ideally encompass an entire digital business flow – the step from Bitcoin to Ethereum.

Meanwhile, in the real world, we have gotten used to unknowable sets of information about us being used in real time to decide whether we should get a job, a lease, a loan, an upgrade or a spouse. No one knows now how many places all over the world may be storing information about them. How many times do we hear as excuse that someone does what he does or is constrained from helping because – the computer.

The mechanisms used to gather data are not necessarily secret and rarely illegal. We consent and do not protest as perhaps, we should, because we accept the business case. Many feel it really doesn’t matter at all. It just happens to be the opposite of informational self–determinism.

3.2.2 The Case for 'Making Blockchains Legal'

It might seem counterintuitive to enhance a technology that was created based on libertarian ideals, such that it can be used to improve mainstream business and the practice of law. This challenge is an important one to pose, also for a plausibility estimate about how fruitful work based on this approach can be expected to be.

The answer is that the Cypherpunks had *agency* in their mind and a *governance* problem to address when they explored decentralization as a solution to the adverse consequences of concentration of power. This proposal is not against the grain of blockchains because it is at its heart concerned with reducing the risk of law enforcement taking blockchain out of business instead of the practice of law being made fairer and more accountable by the application of blockchain. This could be a game changer for the way everyday commerce works, even if it looks paltry in comparison to loftier visions.

[34] 'Cypherpunks' is the name of the members of a mailing list and their heirs, who care deeply about privacy.

[35] Cryptological know-how was export-restricted in the U.S. until the year 2000, some restrictions still remain.

Blockchains also do not make our world more dystopian by introducing more automation. They merely replace centralized mechanisms of control and algorithms used to govern us today with fairer, decentralized, less easily corruptible mechanisms.

3.3 Barrier of Productivity and Readability

The fact that judges and arbitrators cannot currently read smart contracts directly introduces dependency on third parties in their decision process. The way that smart contracts need to be crafted today – see Appendix for source examples – necessarily makes it a collaboration between programmers and lawyers. For several reasons, these professions have a hard time finding a common language, which in turn hampers productivity to the degree that it becomes almost impossible to assure that smart contract code actually expresses what it was intended to do. A communication challenge between professions like this is not a new problem, it exists with every specification effort for a program to be created where developers need to understand what they are asked to build. And it is also inherent to any traditional contract that will express the understanding of the lawyers of what the business people wanted, plus some efforts to protect the latter from themselves. Smart contracts, at this point, suffer both those transitions and the accompanying loss in fidelity to the actual intent.

3.4 Irreversibility

As a function of their capability for fast settlement, currency transfer on a blockchain is irreversible. This is different and very powerful when compared e.g. to payment with credit cards. It protects the merchant – and also criminals – rather than the consumer. It also works without the insurance scheme that underlies the credit card system and reduces costs significantly.

High profile thefts from blockchain accounts, to the tune of tens of millions in fiat, have made obvious that reversibility would be a desirable feature at least for the worst cases. As an exception, the Ethereum community went to great lengths to implement a rewrite of history, the very event that a blockchain is expected to never experience, to revert the theft of $56M from The DAO. Since then, new blockchain projects have been announced that propose a form of democratic governance[36] to reverse criminal acts or grievous errors and technical failure.

[36] Dfinity – https://dfinity.network – and Tezos for example – https://www.tezos.com/static/papers/white_paper.pdf

4 Technical

4.1 Reversibility

4.1.1 The Organic, Next Technology Layer of Blockchains

That a layered architecture should emerge over time for blockchains seems plausible[37] [38] and supported by the way IT develops, including as an example the Internet, which has four communication layers, with each successive one supporting the next higher one and making it possible for it to function.

Irreversibility is an important, enabling feature of the first generation of blockchains that should not be replaced on the foundational level. Reversibility should be introduced as a separate layer that lies between the blockchain system and the application. To stay with the analogy, it might correlate with what *TCP*[39] is for the Internet: unconcerned about the actual application but adding a guarantee to the underlying mechanism that is extended up to the applications.

Reversibility is not the only feature that is needed to make application writing more productive and error-free but as we have seen, it is an indispensable one to make smart contracts ready for business.

4.1.2 Undecided States

Blockchains are no stranger to undecided state. A transaction that has been broadcast has no guarantee that it will ever make it into a block, a circumstance that currently makes application programming on top of blockchains impractical and might stifle the advent of the expected *blockchain killer app*.

Another instance of state oscillation is the *reorg* after a network split[40] – when *proof-of-work*[41] is used – where transactions that looked perfectly fine can be lost without a trace of that they ever existed.

We propose instead that the *business-logic reorganization* that a court decision or arbitration can effect *leave* traces of the now deprecated, 'previous truth' in the system. We further argue that retaining the

[37] James Hazard and Thomas Hardjono predict three layers to emerge for smart contracts, with the proposed language falling into the place of the missing 2nd layer, the "Smart Contracts Description Language: At the middle level a platform-independent language is needed that semantically expresses the identical (equivalent) meaning and intent as captured in the higher level human-readable contract. We refer to this as the Smart Contracts Description Language (SCDL). Such a language must be 'platform-independent', meaning that is must be free for the technical constraints of the target computer or platform (i.e. nodes on the P2P network)." – https://www.w3.org/2016/04/blockchain-workshop/interest/hazard-hardjono.html

[38] The author and Martin Becze predict seven layers of state for a possible new blockchain architecture to allow for 100M transactions per day to reflect the entire business of OTC credit contracts between financial institutions in the Eurozone. Diedrich, Becze, Blockchain Architecture for Supervisory Reporting. Mimeo.

[39] The Internet's **Transmission Control Protocol** provides *reliable*, *ordered*, and *error-checked* delivery of data between applications, building on the Internet Protocol (IP) that does *not* provide these guarantees. The World Wide Web and email are examples for applications that are built on top of TCP. First described in Vinton, Kahn, *A Protocol for Packet Network Intercommunication*, IEEE Transactions on Communications, Vol. 22, No. 5, May 1974 pp. 637–648

[40] A **network split** is the situation that for reasons such as broken cables, a network is split into two or more subgroups that can't communicate with each other for a time or forever. The seminal problem in this case is that the respective groups have no way of knowing whether the participants in the groups that they can't reach anymore have crashed, been shut down, left the network or continue to operate. The problematic phase for most blockchains is the moment that the split ends and the groups have developed different consensus about the world state on their respective sides.

[41] **Proof-of-work** is the notoriously energy-wasting mechanism that public blockchain like Bitcoin and Ethereum use to keep all nodes that participate in the respective network in synch with each other.

previous record will often be indispensable to be able to defend business and legal decisions made on prior, incomplete knowledge.

4.1.3 The DAO: Lock-In and Time Warp

The DAO showed another form of undecidedness, where transactions that were deemed to be criminal were erased from the chain by an act of social consensus, confirming how the agency of humans trumps that of machines for blockchains, too. The DAO also showed the way forward by the importance of the 27 day freeze[42] for the happy end of the story for most[43] investors.

4.1.4 Time Warping Quality of Blockchain Technology

Blockchains *are* very reversible under the hood, e.g. Ethereum is oscillating with regards to what the actual world state should be all the time. The worst type of reversibility of course comes as a price of the consensus algorithm of *proof-of-work* where after a network split, a large minority of nodes[44] might get its entire state wiped and replaced by the evolved opinion of the majority of nodes[45] as soon as the split is overcome.

It is exactly this method of reconciliation that we propose as cornerstone of reversibility, however not on the lower blockchain layer, but on a higher, 'business facts' layer. 'Catching up' is implemented by going back to the last point in time that is not supposed to be reversed and throwing out all changes to the contract's state that had since happened. Then going forward again from there, replaying all transactions up to the present now including the retro-actively added ones that formerly were not part of the list of transactions. These newly added transactions can be inserted before and between the original transactions, independently of when they were actually added, which effects the reversal, whatever form it may take and to whatever outcome it might lead.

4.1.5 The Legal Reorg

This *separation between information and business layer* and the *'reorg' for reversibility on the business layer* – and not at the information layer – is the central tenet of the reversibility mechanism we are proposing. If the reorganization was on the lower, information layer, it would invalidate all cryptographic proofs (hashes) and transactions that happened after the point that is retro-actively deemed to be the point where the reversal should start. This would produce exactly the dreaded effect that a network split can have on a proof-of-work chain and is evidently the wrong place to solve a problem that is really a business logic problem. In the end, judges do not really warp time and they are aware of it. Likewise, we do not want to change the low-level irreversibility and immutability of the blockchain.

[42] The DAO was programmed in such a way, as a security precaution, that funds could not be withdrawn from it for 27 days after certain events. The hack that siphoned $56M out of it triggered such an event and consequently, the thieves could not spend or move the stolen currency for this time. This gave the Ethereum community the chance to engineer the social and technical solution that would 'undo' the entire DAO and with it, the theft of funds out of it. Most investors simply had the currency back in their accounts as if it had never been paid into The DAO.

[43] Some funds had been rescued by different means but could not be restored because the investors could not be identified.

[44] **Nodes** are what the network connects. Usually computers, or devices like mobile phones. For blockchains, they are defined by the piece of software they run, the 'client'. An Ethereum node runs one of the Ethereum client programs, e.g. geth, eth or parity.

[45] More precisely, the majority of calculation power.

4.1.6 Arbitration by Code

Allowing arbitration to not only settle state within the contract but to add code to it, will allow for more satisfactory outcomes in many situations. Such *arbitration by code* mimics the full power that the programers wield, who hack a client to reverse the history of their blockchain. If a simpler approach is preferred, the code added can simply set variables so as to change nothing but state, retro-actively at a certain point in time.

We predict that eventually judges will learn writing code like this as expression of their judgements, after initially using the help of experts to write it. They will be equipped with simulation tools that allow them to test that the concrete outcome or future results of their amendments are indeed what they intend to produce.

4.1.7 Automated Tests

Smart contracts and arbitrations done this way can be tested as to their desired outcome by formal verification methods as well as brute force or Monte Carlo tests. For example, a judge could test their proposed change by running different input parameters that cover important edge cases that may occur going forward from the court decision.

4.1.8 Opcode-level Readability

The representation of programs on the blockchain, the *opcodes*,[46] should allow for unambiguous translation back into the high-level, human readable *source code*[47] to eliminate doubt as to what is actually stored on the blockchain. With blockchains it is not currently a concern that the translation back result into code in a high language that is easy to read for programmers, or that the translation back even be possible at all. To facilitate this, genuine Lexon opcodes should eventually be stored on the blockchain, for the virtual machine to execute. Where *syntactic sugar*[48] is employed, it should be reflected in the opcodes to resolve back unambiguously to make sure that the actual code on the blockchain has exactly one representation when decompiled from opdcodes to the human readable source. *Source readability thus becomes part of the formational criteria that shape the opcode design* for the goal of bi-directional deterministic translation, alongside the expectable goals of performance and conciseness.

As an intermediary step, Lexon can be compiled to other well readable blockchain languages, such as *Ethereum's Solidity*. In fact, we produce an experimental compiler that can do just this. To gain practical use, the Lexon compiler must be held small and easily auditable so that third parties can audit and certify the compiler itself and give testimony that the two outputs are perfectly in sync and express the

[46] **Opcodes** are commands in a language that is easy to understand and fast to execute for machines. It is the form that smart contracts are stored on a blockchain, e.g. with Ethereum. It is usually not possible for humans to write code directly in opcodes, not because they could not learn the commands but because the commands allow only for primitive structuring that would invite errors if humans used them directly to create even moderately complex programs. For how they look, see the EVM opcode example on page 30.

[47] **Source code** is what we mostly look at as programs, a representation that is optimal for programmers to read but not necessarily accessible for the untrained. It is usually not what the machine executes. Source code is often translated by a compiler into opcodes for execution, sometimes *just-in-time*, milliseconds before its execution.

[48] The translation from source code to opcodes usually loses what is called **syntactic sugar**: nuances of expressing logic that makes it easier for humans to read but is unnecessary for the execution by the machine. For example, the ancient programming language *COBOL* has a command *MOVE A TO B*. The *TO* is pure syntactic sugar for human readability. The machine understands what to move to where from the positioning of *A* before *B* and can do with *MOVE A B*.

same logic, one in natural language, one in blockchain smart contract code, for lawyers and judges to firmly rely upon, when crafting or interpreting a contract.

4.1.9 The Mechanism for Reversibility

Reversibility is functionally a temporary change in the burden of proof. Instead of the owner receiving a proof of ownership from the previous owner of a transferred amount, the transferred amount can be deemed to belong to someone else – anyone else – for a predetermined duration. A smart contract's payout is simply kept in escrow by the system itself and the entire contract is suspect to changes by an authorized, freely determined arbitrator until then.

Technically, reversibility is implemented like a chain on top of a chain. The lower, pure information layer records all incoming calls in the order and at the time as received. This is the layer that consensus is formed over. The upper, business facts layer per default is a mirror image of the information layer. But the owner of the arbitration key can, within a predetermined time frame, delete, reorder and add transactions on it.

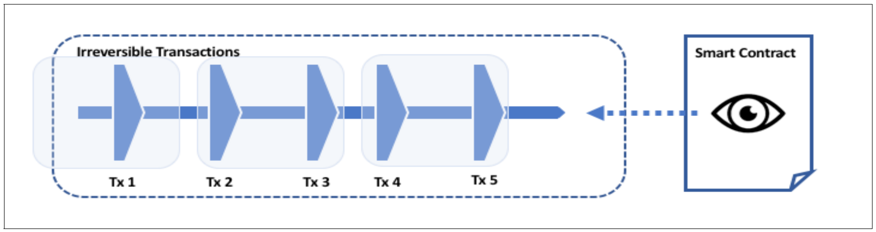

Illustration 1: Normal blockchain with transactions in sequence, smart contract 'looking at' its state, the saldo of all transactions.

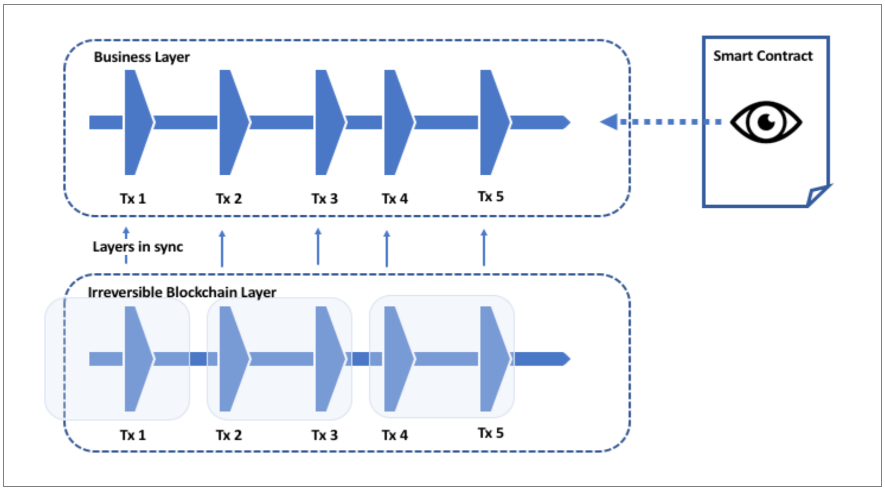

Illustration 2: Two layers: the usual irreversible timestamps (lower) layer and the proposed additional business (higher) layer. In this illustration, still in sync.

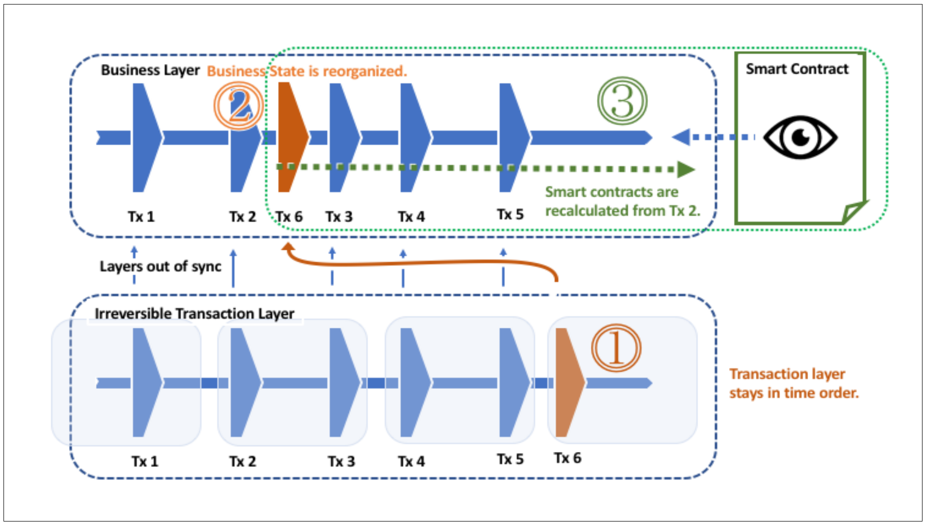
Business Layer
Business State is reorganized.
②
③
Smart Contract
Smart contracts are recalculated from Tx 2.
Tx 1
Tx 2
Tx 6
Tx 3
Tx 4
Tx 5
Layers out of sync
Irreversible Transaction Layer
①
Transaction layer stays in time order.
Tx 1
Tx 2
Tx 3
Tx 4
Tx 5
Tx 6

Illustration 3: Two layers breaking sync: timestamps (lower layer) stay cronological, business layer is reorganized. Resulting state for the smart contract changes. Business and timestamp layer can be translated into each other unambiguously.

In essence, anything can be done to the smart contract that makes it retro-actively behave differently for up to a certain time back that keeps gliding forward. During that gliding window, an arbitrator can change the code of the contract and its payouts are held in an automatic escrow. With respect to the transactions older than the arbitration time window, the smart contract's performances are final. Payouts will have happened and cannot be reversed. A transaction in this context though is any state change within the smart contract, *sometimes* an interaction between accounts. A change in the code of a smart contract is also effected by a transaction.

Reversal works like a blockchain reorg but on the level of business facts only. All consensus and hashes on the information level remain valid. A retro-active change of legal facts, however, e.g. a court order, triggers a new run of the contract logic, from the point in time where the retro-activity is first to be applied, up to the point that it had been executed before. Since now the sequence of events, or the code, or both has been changed, the results might be different, as they should. This happens in a perfectly deterministic way though that does not break consensus between nodes at any time and cannot lead to a schism of the blockchain and its community as witnessed after the Ethereum *hardfork*[49] that reverted The DAO.

The lower-level, information layer works as expected and receives signed transactions that feed state into it. This would usually be signed function calls to smart contracts, complete with specific parameters as payload. These calls are executed as expected and also recorded, with their block height and transaction number relative to the other transactions in the block, in a 'event list'.

4.1.10 The Case for Built-in Reversibility

As an example, say the logic of a contract was (written in logical pseudo code, not Lexon):

[49] See footnote 42.

```
setup(tkn, trs, rcv)
   if no tokens
        tokens = tkn
        trustee = trs
        receiver = rcv

payout(signature)
   if signature is of trustee
        pay tokens to receiver
```

pseudo code (not Lexon)

After the contract is armed by a call to setup that included tokens, it cannot be re-loaded by another call to setup. Specifically, the trustee and receiver cannot be changed, which gives a guarantee of the kind for which smart contracts are made. The caller does not want a possibility that the trustee or receiver could be changed.

However, a court could create a fiction that the trustee, or the receiver, would be different. The reasons for this would usually be from the real world, outside the scope of the blockchain. There is no way to make any contract proof against such eventuality, this is what courts are *for*. Having the ability to change contracts using the court system does not defeat the purpose of contracts. The same is true for smart contracts, having the ability to have them changed in a controlled way by a pre-determined arbitor neither takes away their gains from automation nor general decentralization. Since the court in question is assigned the powers to arbitrate a specific contract, the parties entering the contract have the power to determine who they want to govern the contract and accordingly, which jurisdiction should apply for arbitration.

The owner of the arbitration key of a contract can add and change code of the contract and change the event list of it by a call to it that has an arbitrary block height in the past assigned to it and is inserted in the event list at the position according to its block height.

The owner of the arbitration key of above contract could decide to add a function and call it to set a new receiver:

```
changeReceiver(rcv, signature)
    if signature is arbitrator's
         receiver = rcv
```

example of today's blockchain smart contract, hypothetical pseudo code

The contract could have been implemented with this special arbitration function from the start, except for the two constraining facts that, firstly, bugs happen and there needs to be a way to correct them in a safe way. Secondly, providing for *all* sorts of possible necessities would leave code incredibly cluttered and unreadable. This circles back to our earlier thoughts about Turing-completeness and error-freeness.

The full contract, ready for arbitration *without* reversibility layer would look somewhat like this:

```
setup(tkn, trs, rcv, arb)
    if no tokens
         tokens = tkn
         trustee = trs
         receiver = rcv
         arbitrator = arb

changeTokens(tkn, signature)
    if signature is arbitrator's
         tokens = tkn

changeTrustee(trs, signature)
    if signature is arbitrator's
         trustee = trs

changeReceiver(rcv, signature)
    if signature is arbitrator's
         receiver = rcv

payout(signature)
    if signature is of trustee
         pay tokens to receiver
```

example of today's blockchain smart contract, hypothetical pseudo code

It's obvious how one could wrap all state in arbitration setters like this, and that this has uncanny likeness with error handling without *exceptions*[50] or with an attempt to implement *transaction handling*[51]

[50] **Exceptions** are a programming device whereby a program can be instructed to just bail and somewhat brutally jump back to a prior set safe place, abandoning the current execution path completely. It saves the normal program flow from being cluttered by error handling code that obscures the actual program logic. It can also 'forget' a lot of state that was created along the way from the safe place to when the exception was triggered and produces a somewhat clean slate.

[51] **Transactions** are a decades old. elemental concept for database programming. They are indispensable for programming safe double bookkeeping in that they make sure that transacted numbers cannot inadvertently be *lost* nor *doubled* by a system crash. For example, if 10 units were taken from account *A* to be transferred to account *B*, in a database scenario they have to be subtracted from account *A* and added to account *B*. The *transaction* mechanism allows a programmer to declare that either both or neither of the subtracting and adding should happen, even in the event of a crash half way. The programmer can also decide to *abort* a transaction, for example if it had been found that the account *B* was deleted for some unknown reason just the moment after the subtraction of 10 units from *A* but before adding them to *B*. For a simple example the advantage of the ensuing *roll back* that restores the subtracted 10 units to *A* automatically is not obvious but transactions can be complex and involve multiple conditional changes to data points, all of which can be safely rolled back with one command, which makes transaction abortion similar but even more powerful than an *exception.* If a database, such as some of the newer *key-value stores* does not extend a mechanism for transactions to the programmer, it can be impossible to create a safe substitute on the application level. Safe transactions then might simply be off limits. But note that for many applications, for example for the implementation of web shopping carts, they are not needed. Transactions in this sense were described by Jim Gray in 1981: Gray, *The Transaction Concept: Virtues and Limitations*. Proceedings of the 7th International Conference on Very Large Databases. Cupertino, CA: Tandem Computers. pp. 144–154 – https://pdfs.semanticscholar.org/fff4/0975df8dfc429061930f91079ea87a02352b.pdf

without *ACID*[52] guarantees from the database layer. An error *in* the arbitration functions could still *not* be corrected. Furthermore every single added function expands the *attack surface*[53] and should better only exist when absolutely needed. The arbitration functions are also obviously boilerplate that do not add core business functionality per se, and would be best generated by the editor or compiler. Better yet, they should be provided on the proposed at-need basis by the underlying system.

4.1.11 Ripple Effects

The proposed reversion layer goes much further. It could for example just take out the payout function as if it never existed, even after the payout happened and create a refund function and call that instead. Functionality-wise, this now adds this code to our initially short smart contract:

```
refund(rcv, signature)
    if signature is arbitrator's
          receiver = rcv

payout(signature)
    if signature is of trustee
          escrow = tokens
          tokens = 0

release(signature)
    if signature is arbitrator's and arbitration over
          pay escrow to receiver
```

example for verbosity of today's blockchain smart contract, hypothetical pseudo code

4.1.12 Fractional Reserve

The automatic escrow we are proposing can be extended into a fractional reserve system that allows for uncovered transactions that might be clawed back at a later time, extending ripple effects into the system for such unsecured tokens.

[52] **ACID** is the description of what intuitively would be expected from a database: that it stores what it is given and returns it later. Meaningful *transactions* are only possible when a database offers ACID guarantees. That is technically not trivial however and even high-end databases like to fudge parts of it to gain on performance. ACID stands for *atomicity*, *consistency*, *isolation* and *duration*. It was coined by Härder, Reuter, 1983, *Principles of transaction-oriented database recovery,* ACM Computing Surveys – https://www.researchgate.net/publication/220566390

[53] The **attack surface** is the breadth of options that a potential attacker has. It is the broader and thus the system the more vulnerable, the more different ways to access it exist. For example, the *parity wallet hack* that caused damage of $32M was caused by hackers exploiting an opening that had been exposed presumably for appropriate and important testing purposes but unintentionally left open when commissioning the wallet software into service. It was one function too much, similar to how we are adding functions in our arbitration example at hand.

4.2 Language

4.2.1 Token Example

The following is an example of the new language in low level application. It defines a standard token.

```
1.     ARTICLE: Token.
2.
3.     COMMENT: array for the token balances.
4.     Balance: map id to amount.
5.
6.     PARAMETERS:
7.     Comment: initialize contract with tokens supply.
8.     Initial Supply: amount.
9.
10.    PREAMBLE:
11.    Comment: give the creator all initial tokens.
12.    Balance of Creator be Initial Supply.
13.
14.    CHAPTER: transfer Amount from Sender Id to Receiver Id.
15.    Provided: Balance of Sender greater or equal Amount.
16.    Provided: Balance of Receiver added to Amount
17.              greater Balance of Receiver.
18.    Subtract Amount from Balance of Sender.
19.    Add Amount to Balance of Receiver.
```

Lexon source code

Line 1 uses the keyword *article* to start a library entry. It is called 'Token'.

Line 3 is a comment that is private to the programming side and will not appear in the prose text.

Line 4 defines a variable, named 'Balance', and its type: being a map of ids that point to amounts. All of 'map', 'id' and 'amount' are built-in, primitive types. 'id' is a type that takes integers, hashes or public keys. 'amount' is a type for currency.

Line 8 defines a parameter called 'Initial Supply', which is to mean the totality of all tokens that this contract will manage and that is set when the contract is first setup. The line also defines that 'Initial Supply' is a variable of the type 'amount'.

Line 12 declares that the Balance of the Receiver is initially to be the entire initial supply of tokens.

Line 14 defines the action of transferring tokens, in the instance this is a *method signature*: its name being 'transfer', its first parameter of type *and* name 'amount', its second parameter the Sender ID, the third the Receiver ID. The words 'from' and 'to' are syntactic sugar that is used to create a better human readable output (see '1st', below).

Line 15 and 16 are assertions: if the conditions stated after the keyword 'Provided' are not met, the interpretation of the chapter ends, i.e. an exception is thrown and the rest of the code is ignored for this call.

Line 17 and 18 get to the heart of the matter and transfer the tokens out of the Sender's account into the Receiver's.

Our experimental compiler produces *both* the following outcomes from the above code.

```
CHAPTER 1: The Token.
The Balance is defined as a ledger that has one amount entry
for every identity. Each entry is initially 0.

PREAMBLE.
When this contract offering is first created, decisions are
made by the creator regarding the offering, as follows:
The Initial Supply is set to a specific amount.
The Balance of the Creator is set to equal the Initial Supply.

Clause 1: Transfer an Amount From a Sender To a Receiver.
Provided that, the Balance of the Sender is greater than or
equal the Amount.
Provided that, the result of the Balance of the Receiver added
to the Amount is greater than the Balance of the Receiver.
The Amount is subtracted from the Balance of the Sender.
The Amount is added to the Balance of the Receiver.
```

1st product: Human readable contract prose.

First a human readable text that truly reflects what the *second* product (on the next page) does: a Solidity source for execution on Ethereum.

```
pragma solidity ^0.4.0;

contract token {
     /* array for the token balances */
     mapping (address => uint256) balance;
     event Log(
          bytes32 indexed _hash,
          string _value
     );

     /* initialize contract with tokens supply */
     function token (uint256 initial_supply) {
          /* give the creator all initial tokens */
          balance[creator] = initial_supply;
     }

     function transfer(uint256 amount, address sender,
                         address receiver) {
          if (!(balance[sender] >= amount) {
              throw();
          }
          if (!(balance[receiver] + amount > balance[receiver]) {
              throw();
          }
          balance[sender] -= amount;
          balance[receiver] += amount;
     }
}
```

2nd product: Ethereum–ready smart contract source.

Both outputs mean the same but are optimized, respectively, for human readability or machine readability. The above is not a theory but output from our experimental compiler when fed with the Lexon code as shown above.

4.2.2 Structure

The language, visually and structurally, follows a very simple pattern, namely that of defining a name. This is the nucleus of most of its statements:

name: definition.

To keep the optics simple and closer to natural flow of thought and language, as a guiding star, anything that would required nested structures is instead required to be factored out into independent blocks. So there are no nested if-clauses beyond two levels, instead the blocks have to be encapsulated into sub functions.

The keywords are not case sensitive. Variables are typed, can have spaces in their names and be named after a type, in which case the declaration is shorter. Indentation and line-endings do not carry meaning. As opposed to pure functional notation, the order of lines is relevant.

The overall structure is as follows:

- A *Contract* is a list of *Articles*.
- An article starts with a symbol that defines it and a list of parameters that represent the input to the source code of the article that is expected at run time of the contract. After a colon, a list of *statements* follow.
- The statements are conditional expressions or assignments of values to variables that will control what other statements will be relevant and what the contract might result into.

- The statements can be sub grouped into chapters, which have names, too, but no parameters. They otherwise also consist of a list of statements.
- The contract is the overarching bracket that holds the application. The articles are obviously the 'functions' or 'methods' that can be called from the outside to trigger a state transition of the contract. The chapters are important to give the statements structure, particularly to be able to nest deeper if-else structures.
- Any name can be defined by writing it out and adding a colon. This is also the way to declare variables, in which case a type and a period is all that follows.
- The keyword *Provided* is an assertion and if the condition following does not evaluate to true, then the article's remaining statements are ignored.
- Lines that start with the keyword *Comment* are ignored.

4.2.3 Formal Grammar

This is the formal language grammar as currently developed.

```
contract: article_list

article_list: article
            | article_list article

article: ARTICLE ':' SYMBOL '.' stmt_list PARAMETERS ':'
          parameters CONSTRUCTOR ':' stmt_list chapters
            | ARTICLE ':' SYMBOL '.' stmt_list chapters
            | ARTICLE ':' SYMBOL '.' chapters

chapters: chapter
            | chapters chapter

chapter: CHAPTER ':' signature '.' stmt_list

signature: symbol
            | symbol arguments

arguments: argument
            | arguments filler argument

argument: type
            | symbol typedef

filler: TO
            | FROM
            | /* empty */
```

```
parameters: commented_parameter '.'
            | parameters commented_parameter '.'

commented_parameter: parameter
            | comment '.' parameter

parameter: symbol ':' typedef

comment: COMMENT

definition: symbol ':' typedef

provision: PROVIDED ':' boolexpr '.'

typedef: MAP type TO type
            | type

type: ID
            | AMOUNT

symbol: SYMBOL

stmt: '.'
            | expr '.'
            | definition '.'
            | provision
            | LOG expr '.'
            | VARIABLE '=' expr '.'
            | lval '=' expr '.'
            | WHILE expr ':' stmt
            | IF expr ':' stmt
            | IF expr ':' stmt ELSE stmt
            | '(' stmt_list ')'
            | COMMENT

stmt_list: stmt_list stmt
            | stmt

boolexpr: expr '<' expr
            | expr '>' expr
            | expr GE expr
            | expr LE expr
            | expr NE expr
            | expr EQ expr
            | '(' boolexpr ')'

expr: INTEGER
            | VARIABLE
            | type
            | '-' expr
```

```
             | expr ADDED filler expr
             | expr '+' expr
             | expr '-' expr
             | SUBTRACT expr FROM lval
             | ADD expr TO lval
             | expr '*' expr
             | expr '/' expr
             | boolexpr
             | '(' expr ')'
             | lval

lval: symbol
             | symbol OF symbol
```

4.2.4 Definable Grammar

It is possible that the ideal compiling process would be to make two full cycles (not passes), the first one of which would allow to define grammar, specifically to describe new syntactic sugar, while the second applied this grammar to the remaining source. This would mean to first extract grammar input to the compiler compiler (sic) to build the compiler that then compiles the code. This is not how usual programming languages work that have a fix grammar.[54] It might be desirable in this case to be able to stay closer to human language and optimize the human language output for readability.

Adding new syntactic sugar would mostly happen when defining frameworks rather than real contracts. For example, the token example above could fix as requirement, the *From* and *To* in the line *Transfer an Amount From a Sender To a Receiver.*

4.2.5 Multi-lingual Grammars

The proposed language should have different versions for different languages and jurisdictions. The principles have been shown to work for German and English. There might be hard problems with natural languages that have a radically different structure.

5 The Way Forward

5.1 Coded Law

The question that inevitably fires in the mind of every interested lawyer is: *can all law and regulatory texts at one point be expressed in a smart contract-like language?* There are reasons why IT[55] system developers might refrain from even asking this.

[54] The **Grammars** of computer languages are not of the same category as the grammars of human languages. But human languages obviously have mostly fix grammars, too. The fact that the definition of grammar might figure in the compilation process is a reflection of the fact that our proposed language does *not* (and is not supposed to) really 'understand' human language grammar, it merely mimics it.

[55] **Information Technology** – all things computer.

5.1.1 Approach

In our view, the prize of creating a machine-processable mirror image of laws[56] that will allow for fully automated contract performance and AI[57] judges can be won by working first from the ground up, focussing on blockchain smart contracts, giving them a path to legal correctness. This establishes the channel between the sufficiently different universes of law and computer code and it is likely that the portal once opened can then be used in the other direction.

To find the way in the reverse direction first,[58] trying to make *law smart* rather than *smart lawful* is, by definition, a task that is so maximally generic that it will suffer from the lack of use-cases or in the best case, from failure to manage the vast array of them.

A meaningful reflection about expressing *all* of law and regulations in a language like that proposed here for smart contracts, can start with the question: *What algorithms would be needed and how do we supply facts to create a framework that a smart contract can be plugged into for it to be supplied with parameters and behavior that allows it to cover many edge cases implicitly* that one would not spell out in a contract on paper because one knows that they are covered by the contract law of the land? Of course, different jurisdictions would supply different frameworks and accordingly, see the contract behave differently.

5.1.2 Feasibility

Since all regulations are input to a potential decision making process to determine if something is allowable or not, all regulations should be expressible as frameworks for actual contracts, with the expected analogy with OO frameworks that there is no actual processing code, just abstract frame that must be brought to life by a concrete instantiation of a contract or a case.

Law, especially Civil Law, as an even more abstract source of rules for concrete contracts should be expressible in the same way and we have described how *defining* grammar, as opposed to only *using* it in normal programming languages, might become the hallmark of the very high abstraction of the task we are looking at.

While regulations tend to have concrete figures as part of their prose, law texts rarely do and their required terseness and generality point to how they are really models of reasoning, defining the structures that have to be evoked as relevant by finding what concrete facts of a case fit what definition, or what precedence implies a pattern that can be argued to be a match for a case at hand. Unsurprisingly, different cases can be made based on different interpretation of facts using different aspects of the definitions.

It remains to be seen to what extent this act of matching definitions and patterns can ever be automated. On the face of it, the overlap with the very core promises of AI is uncanny and a majority of court cases

[56] Casey, Niblett, *The Death of Rules and Standards:* “As standards disappear and judges have progressively less influence, legislative intent will be entrenched and concretized in the catalog of micro-directives. Technological changes that vastly improve ex ante information will also breathe new life into old law-and-economics models that began with an assumption that lawmakers and citizens have full information.” – https://mirror.explodie.org/The%20Death%20of%20Rules%20and%20Standards.pdf

[57] **Artificial Intelligence** – the voice in an elevator might have passed as AI in the 30's. Now we are more spoiled but still won't expect AI, even when used for sentencing, to really be sentient. Just incredibly adept at research into precedence and flawless in applying rules.

[58] Meng Wong is a champion of the practical work in this field and his website the ultimate treasure trove of information about it: http://legalese.com

– the simpler as well as the very convoluted ones – might see at least support for the judge by AI that taps into structured law.

5.1.3 Maintenance

Another concrete challenge lies in the fact that large systems become unwieldy. A common problem in programming is that large frameworks require intricate setups and have to make assumptions to allow for a smooth ride of their payload, the concrete program at hand. These assumptions can be wrong and cause hard to find bugs. In the same vein, very many lines of code in a framework are often executed for no good reason other than that it had not been optimized to leave out parts that are not needed for one concrete case, or because optimization poses a hard problem that is expensive to solve. The unneeded lines of codes will also rest on inputs and so might require input that is actually not really required except to be able to run these – in the instance – unneeded code parts. The required decisions here might be very easy for human beings but impossible for a program. Employing AI can be a solution but it is the kind of AI that needs to be trained and brings with it, for this reason, fallibility.

For a framework of law and regulations this indicates that a major challenge will lie in finding out which facts need to be established and which are irrelevant, so as to execute as little code as possible on every execution. This is the most abstract requirement because it asks *reflection* of the system, a notoriously expensive feature that can lead to bad code that is hard to maintain and hold error-free when used wrongly.

In the end, many facts might routinely have to be established with authority to employ a vast, automated regulatory and law frameworks, even to execute only tiny contract scripts. While this might now look unlikely it may simply lead to a new industry similar to the tasks of notaries, to supply facts to the system, possibly on a ad-hoc basis, as so called 'oracles'. Once established, facts can be re-used like precedence. But different to the sort of law we are used to, the new one would require some sort of upkeep.

5.2 Governance

The purpose of this proposal is to enable a formidable leap of the practice of law to the better. This cannot be achieved without the collaboration of dedicated specialists on both sides of the divide between technology and law. While the goal of the new language is to allow better communication between lawyers and technicians, a lot of communication must happen to make it real in the first place.

The process should be guided by collectives that give themselves a form suitable to reaching the goals laid out above and guard a sustained effort that can politically survive pivots where found necessary. The outcomes must be seen as a common good.

5.2.1 Common Dictionaries

Each law firm could create their own libraries of legal code, which is in fact how many small firms have carved out their niches using manual word processing, by perpetually modifying previously crafted contracts, using prior work as template. Common dictionaries of processable law can have impact of a different magnitude however, transforming the way law is practiced.

The true game changer will lie in libraries that emerge from the joint effort of specialists to transfer legal texts into processable formats. This will likely change the legal professions in profound ways as a lot of grind is handed over to the machines, while eliminating many potentials for rent seeking. The incentives for the collaborators in this endeavor must therefor be clarified.

5.2.2 An Academy of Regulations and Law

To begin in earnest the Herculean task to transfer laws into code that is machine-processable, governmental committees on the highest, even supra-national level must be formed by members of agencies and government bodies that deal in such topics. An Academy, modeled after the Académie française,[59] that brings together specialists and lawmakers, for example on the European level, could begin to define the vocabulary needed to codify law in a new way. In the U.S., the American Law Institute[60] could create a new 'project' dedicated to this cause.

This will complement the grass roots approach of making blockchain smart contracts legally correct that will allow people to experience what is possible and establish a new way of thinking about laws and regulations.

[59] The **Académie française**, established in 1635, is the pre-eminent council for matters pertaining to the French language, giving advise about grammar, spelling, and literature. The Académie consists of forty members, known as *les immortels*. It was modeled after the *Accademia della Crusca,* the oldest linguistic academy in the world and the first academy devoted to eliminating the "impurities" of a language. Founded in Florence in 1582, the Accademia secured the ascent of the dialect of Florence to the common Italian language. The Accademia published its *Vocabolario* after thirty years of work in 1612. The *Oxford English Dictionary* took 71 years to produce, the Grimm's *Deutsches Wörterbuch*, 123 years. The vision of the European Academy for Processible Law has been articulated by Harald Stieber.

[60] The American Law Institute is “the leading independent organization in the United States producing scholarly work to clarify, modernize, and otherwise improve the law.”

6 Appendix: Earlier Smart Contract Languages

In the following we give short examples of the optics of blockchain languages as they exist today, which no[61] practicing lawyer will bother to try to decipher. Some are cryptic even for programmers, some look like, some are, existing mainstream languages and thus very accessible for programmers.

What is striking is how much very low level processing is handled in the contract code examples and how that obscures the actual business logic that should be the focus of a contract script.

We will go from low to high-level languages, which will also demonstrate again that programming languages are all about readability for humans and smart contract languages have come a long way already. We are proposing but the ultimate step.

6.1 Bitcoin

Script

This is a simple transaction of bitcoins in the opcodes of Bitcoin's virtual machine, called *Script*.

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

There is no expectation that anyone would program applications directly in Script. Rather, there are a handful of standard transactions that a *wallet*[62] needs to be able to construct, which it does by filling in public keys and signatures at the appropriate places.

Script serves to allow for a very safe and convenient way to create more interesting types of transactions. It's use is severely restricted for security reasons as most miners will only allow six different Script programs to be used for transacting. However, it serves its purpose well and new types of transactions can and are programmed in it and approved as standard by the Bitcoin community when found sensible and safe. Similar to Ethereum, allowing new Script transaction types into the standard does not invalidate earlier blocks or require client updates.

6.2 Ethereum

EVM

The Ethereum Virtual Machine (EVM) was built for more complex monetary transactions. A dump of its low-level opcodes looks as cryptic as Bitcoin transactions. It is *not* made to be programmed directly, but as a target for the five languages compiling to it: *Solidity*, *Viper*, *Serpent*, *LLL, Mutan*. These are all

[61] From anecdotal evidence in a group of very blockchain-affine law professionals.

[62] Wallets are Bitcoin clients for users that show a users owned total of Bitcoins and allows the user to make transactions to other users' accounts.

high languages invented specifically for Ethereum but inspired by *JavaScript*, *Python*, *again Python, Lisp* and *Go,* respectively.

The EVM is today's de-facto standard as blockchain virtual machine. *Ethereum*, *Tendermint* and *Rootstock* all use the EVM. *Hyperledger* is looking into utilizing it. *Dfinity* was planning to use it. *Monax* had forked it but made an effort to get back to using the original one.

```
PUSH1, 67, DUP1, PUSH1, 11, PUSH1, 0, CODECOPY, PUSH1, 78,
JUMP, PUSH29, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, PUSH1, 0, CALLDATALOAD,
DIV, PUSH4, 238, 233, 114, 6, DUP2, EQ, ISZERO, PUSH1, 65,
JUMPI, PUSH1, 4, CALLDATALOAD, PUSH1, 64, MSTORE, PUSH1, 2,
PUSH1, 64, MLOAD, MUL, PUSH1, 96, MSTORE, PUSH1, 32, PUSH1,
96, RETURN, JUMPDEST, POP, JUMPDEST, PUSH1, 0, RETURN
```

LLL

The Lisp-like LLL is hoped to be easier to verify for correctness than the other languages that cater to the EVM. Monax liked using LLL when they started out with Eris but are now basing their contract framework on Solidity. LLL won a new lease on life, however, when the core of the very important effort to give Ethereum a name service was ported from Solidity to LLL in a bid to make the code more secure and faster, because LLL can be easier to review and the compiler produces less opcode compared to Solidity. This has lead to bold calls for LLL’s ‘resurrection’ on the grounds that 'smart contracts are hard at any rate', to say that: making them easy to program by having an easy to read and write language for them, was leading into the wrong direction.

Note that there are enough programmers who find Lisp-style elegant and easier to read and reason about than other languages.

```
return 0
  (lll
    (with '__funid
      (div (calldataload 0)
        26959946667150639794667015087019630673637144422540572481103610249216
      )
      (unless (iszero (eq (get '__funid) 4008276486))
        (seq
          (set 'x (calldataload 4))
          (seq
            (set '_temp_521 (mul (get 'x) 2))
            (return (ref '_temp_521) 32)
          )
        )
      )
    )
    0
  )
)
```

Serpent

Serpent is a language that Vitalik Buterin liked to use to try new things in for Ethereum, before he developed *Viper*. *Serpent* was the first high language of *Ethereum* and allows for a more direct influence on the resulting opcodes going to the virtual machine than *Solidity*. This example is a one-time writable key-value store, like a first-come name service.

```
def register(key, value):
   # Key not yet claimed
   if not self.storage[key]:
        self.storage[key] = value
        return(1)
   # Key already claimed
   else:
        return(0)

def ask(key):
    return(self.storage[key])
```

Solidity

Solidity is a Javascript-like, general purpose language designed for programming decentralized applications, but also the best compensation for some deficiencies of the Ethereum virtual machine.\J If anyone talks about programming Ethereum, they will usually mean, programming Solidity. The development ecosystem springing up around Ethereum is almost exclusively geared towards Solidity and it's the most sophisticated compiler for Ethereum.

In a way Solidity is a good as it gets these days. The non-Ethereum examples below all suffer from a less successful separation of concern where system overhead has seeped (or is still part of) the code of the smart program when it should actually express higher lever, business logic. This goes back to *before* Bitcoin where the insulation of the transaction code from the underpinning mechanisms of e.g. consensus and storage was the very reason that the virtual machine was implemented.

```
//sovereign currency parameters of a given Community
contract sovereign {
   address Treasury; //the address of the Treasury of the DAO.
   address Commons;  //the address of the Community account.  int de-
murrage;    //the depreciation at each transaction.
   uint reward;      //reward to the moneyLender of a credit
   int available;    //the spending limit of an account
   int amount;
   int creditLine;

   struct sovereignWallet {
         int sovereigns;      //actual balance in a wallet
         uint credit;         //negative limit
         uint deadline;       //limit of creditline
```

```
          address moneyLender; //the credit line authorizer
          uint reputation;     //available reputation cost
          uint reputationCost; //cost of a credit authorization
    }

    function sovereign() {
          Treasury = msg.sender; //address of the delegated DAO
          Commons = 0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae;
          //the address of the DAO to manage public accounts.
          demurrage = 1;
          reward = 20;
    }

    mapping (address => sovereignWallet) balances;

    //Treasury can issue any amount of sovereign or reputation
    function createAssignSovereigns (address beneficiary,
          int funds) {
         if (msg.sender != Treasury) return;
          balances[beneficiary].sovereigns += funds;
    }

    function createAssignReputation (address beneficiary,
          uint funds) {
         if (msg.sender != Treasury) return;
         balances[beneficiary].reputation += funds;
    }

    //function make a payment
    function pay(address payee, uint payment) {
          //update the credit status
          if (balances[msg.sender].credit > 0) {
              //check if deadline is over
              if (block.number > balances[msg.sender].deadline) {
                 //credit was not returned
                 if (balances[msg.sender].sovereigns < 0) {
                       //lender reputation not restored
                       //and penalized with 20%
                       balances[balances[msg.sender].
                             moneyLender].reputation -=
                       balances[msg.sender].reputationCost
                       * reward/100;
                 }
                 //credit was returned
                 else {
                       //money lender reputation is restored
                       //and rewarded with a 20%
                       balances[balances[msg.sender].
                             moneyLender].reputation +=
                       balances[msg.sender].reputationCost
                       * (100 + reward)/100;
                       //reset credit to zero, deadline to zero
                       balances[msg.sender].credit = 0;
                       balances[msg.sender].deadline = 0;
                 }
          //if time is not over proceed with the payment
          }
    //if there was no credit proceed
    }

    //pay with the reviewed balance and credit
    creditLine = int(balances[msg.sender].credit);
    available = balances[msg.sender].sovereigns + creditLine;
    amount = int(payment);
    if (available > amount) {
          balances[msg.sender].sovereigns += amount;
          balances[payee].sovereigns += amount;
          //apply demurrage
          balances[payee].sovereigns -=
              amount * (100 - demurrage)/100;
          balances[Commons].sovereigns +=
              amount * (demurrage)/100;
    }
 }

 //authorize a credit
 function credit(address borrower, uint credit, uint blocks)
    returns(bool successful) {
    if (balances[msg.sender].reputation > credit * blocks) {
              balances[msg.sender].reputation -= credit * blocks;
              balances[borrower].credit += credit;
              balances[borrower].moneyLender = msg.sender;
              balances[borrower].deadline =
                 block.number + blocks;
              //deadline in blocks ahead
              balances[borrower].reputationCost =
                 credit * blocks;
        }
    }
 }
```

6.3 Corda

KOTLIN

Corda, the project of the banking consortium *R3 CEV*, is not a blockchain but a *message passing architecture* and the smart contracts it claims to have are inspired by a not too closely related name-sake, *Barclay's Smart Contract Templates* that clearly define themselves as *not* smart contracts in the sense commonly understood in connection with blockchins. But Corda is of course also concerned with automation of transactions and its business logic source is programmed in Kotlin, a new variant of Java. This is a great choice for reaching as broad a programmer community as possible and *Dfinity* will target Java as its native programming language, too.

The actual code covers a lot of rather deep down, technical communication elements that have little to do with the level of the business logic, which in this case is simple enough: the issuance of an amount of cash by a bank.

```
package net.corda.bank

import com.google.common.net.HostAndPort
import joptsimple.OptionParser
import net.corda.bank.api.BankOfCordaClientApi
import net.corda.bank.api.BankOfCordaWebApi.IssueRequestParams
import net.corda.flows.IssuerFlow
import net.corda.core.node.services.ServiceInfo
import net.corda.core.node.services.ServiceType
import net.corda.core.transactions.SignedTransaction
import net.corda.flows.CashFlow
import net.corda.node.driver.driver
import net.corda.node.services.User
import net.corda.node.services.startFlowPermission
import net.corda.node.services.transactions.SimpleNotaryService
import kotlin.system.exitProcess

/**
 * This entry point allows for command line running of the
Bank of Corda functions on nodes started by BankOfCordaDriver.kt.
 */
fun main(args: Array<String>) {
    BankOfCordaDriver().main(args)
}

private class BankOfCordaDriver {
    enum class Role {
        ISSUE_CASH_RPC,
        ISSUE_CASH_WEB,
        ISSUER
    }

    fun main(args: Array<String>) {
        val parser = OptionParser()
        val roleArg =
parser.accepts("role").withRequiredArg().ofType(Role::class.java).describedAs("[ISSUER|ISSUE_CASH_RPC|ISSUE_CASH_WEB]")
        val quantity = parser.accepts("quantity").withOptionalArg().ofType(Long::class.java)
        val currency = parser.accepts("currency").withOptionalArg().ofType(String::class.java).describedAs("[GBP|USD|CHF|
EUR]")
        val options = try {
            parser.parse(*args)
        } catch (e: Exception) {
            println(e.message)
            printHelp(parser)
            exitProcess(1)
        }

        // What happens next depends on the role.
        // The ISSUER will launch a Bank of Corda node
        // The ISSUE_CASH will request some Cash from the ISSUER on behalf of Big Corporation node
        val role = options.valueOf(roleArg)!!
        if (role == Role.ISSUER) {
            driver(dsl = {
                val user = User("user1", "test", permissions = setOf(startFlowPermission<CashFlow>(), startFlowPermission<IssuerFlow.IssuanceRequester>()))
                startNode("Notary", setOf(ServiceInfo(SimpleNotaryService.type)))
                startNode("BankOfCorda", rpcUsers = listOf(user), advertisedServices = setOf(ServiceInfo(ServiceType.corda.getSubType("issuer"))))
                startNode("BigCorporation", rpcUsers = listOf(user))
                waitForAllNodesToFinish()
            }, isDebug = true)
        }
        else {
            try {
                val requestParams = IssueRequestParams(options.valueOf(quantity), options.valueOf(currency), "BigCorporation", "1", "BankOfCorda")
                when (role) {
                    Role.ISSUE_CASH_RPC -> {
                        println("Requesting Cash via RPC ...")
                        val result = BankOfCordaClientApi(HostAndPort.fromString("localhost:10004")).requestRPCIssue(requestParams)
                        if (result is SignedTransaction)
                            println("Success!! You transaction receipt is ${result.tx.id}")
                    }
                    Role.ISSUE_CASH_WEB -> {
                        println("Requesting Cash via Web ...")
                        val result = BankOfCordaClientApi(HostAndPort.fromString("localhost:10005")).requestWebIssue(requestParams)
                        if (result)
                            println("Successfully processed Cash Issue request")
                    }
                    Role.ISSUER -> {}
                }
            }
            catch (e: Exception) {
                printHelp(parser)
            }
        }
    }
}
```

6.4 RChain

Rholang

Rholang[63] is the language that Greg Meredith is developing for his next-generation blockchain *Rchain*. The chain is described as solving the challenges of scalability and the language is most fascinating, based on the *rho calculus* for expressing parallel processes, but maybe not made for non-programmers.

This is an example that sets or reads out one value in a perfectly parallelized way.

```
contract Cell( get, set, state ) = {
  select {
    case rtn <- get; v <- state => {
     rtn!( v ) | state!( v ) | Cell( get, set, state )
    }
    case newValue <- set; v <- state => {
      state!( newValue ) | Cell( get, set, state )
    }
  }
}
```

6.5 Hyperledger

Go

Hyperledger, the blockchain by IBM, is programmed in *Go*, which in theory makes it very accessible to programmers, if less so for legal experts, Go being an elegant but complex language. In practice, the business logic of a IBM's 'chaincode' is hard to follow and express for programmers, too, because basic functionality from the underlying ledger system has to be dealt with in a verbose manner on the application level, i.e. in the Go code of the chaincode as shown below. This is a development in the opposite direction from what we are proposing and a good example of the negative impact on readability when concerns have to be addressed on the wrong layer, too far up, because the lower layer is not providing them. In this case, IBM's developers are using the original Go compiler to create the smart contracts that does not provide the basic support that e.g. the EVM extends for a smart contract behind the scenes.

```
// ************************************
// updateAsset
// ************************************
func (t *SimpleChaincode) updateAsset(stub *shim.ChaincodeStub, args
[]string) ([]byte, error) {
   var assetID string
   var argsMap ArgsMap
   var event interface{}
```

[63] Example from: https://raw.githubusercontent.com/rchain/Rholang/DAO/examples/sugar/Cell2.rho

```
  var ledgerMap ArgsMap
  var ledgerBytes interface{}
  var found bool
  var err error
    var timeIn time.Time

  if len(args) != 1 {
       return nil, err
  }

    assetID = ""
    eventBytes := []byte(args[0])

    err = json.Unmarshal(eventBytes, &event)
    if err != nil {
        return nil, err
    }
    if event == nil {
       return nil, err
    }

    argsMap, found = event.(map[string]interface{})
    if !found {
        return nil, err
    }

    // is assetID present or blank?
    assetIDBytes, found := getObject(argsMap, ASSETID)
    if found {
        assetID, found = assetIDBytes.(string)
        if !found || assetID == "" {
            return nil, err
        }
    }

    found = assetIsActive(stub, assetID)
    if !found {
        // redirect to createAsset with same parameter list
        if canCreateOnUpdate(stub) {
            var newArgs = []string{args[0], "updateAsset"}
            return t.createAsset(stub, newArgs)
        }
        return nil, err
    }

    // test and set timestamp
    // TODO get time from the shim as soon as they support it, we
cannot
    // get consensus now because the timestamp is different on all
peers.
   //***************************************************//
    // Suma quick fix for timestamp  - Aug 1
    var timeOut = time.Now() // temp initialization of time variable
- not really needed.. keeping old line
    timeInBytes, found := getObject(argsMap, TIMESTAMP)

    if found {
        timeIn, found = timeInBytes.(time.Time)
        if found && !timeIn.IsZero() {
            timeOut = timeIn
        }
    }

    txnunixtime, err := stub.GetTxTimestamp()
   if err != nil {
        return nil, err
   }
    txntimestamp := time.Unix(txnunixtime.Seconds, int64(txnunix-
time.Nanos))
    timeOut = txntimestamp
    //***********************************************//
    argsMap[TIMESTAMP] = timeOut
    // *********************************
    // find the asset state in the ledger
    // *********************************
    log.Infof("updateAsset: retrieving asset %s state from ledger",
assetID)
    assetBytes, err := stub.GetState(assetID)
    if err != nil {
        log.Errorf("updateAsset assetID %s GETSTATE failed: %s", as-
setID, err)
        return nil, err
    }

    // unmarshal the existing state from the ledger to theinterface
    err = json.Unmarshal(assetBytes, &ledgerBytes)
    if err != nil {
        log.Errorf("updateAsset assetID %s unmarshal failed: %s",
assetID, err)
        return nil, err
    }

    // assert the existing state as a map
    ledgerMap, found = ledgerBytes.(map[string]interface{})
    if !found {
        log.Errorf("updateAsset assetID %s LEDGER state is not a map
shape", assetID)
        return nil, err
    }

    // now add incoming map values to existing state to merge them
    // this contract respects the fact that updateAsset can accept a
partial state
    // as the moral equivalent of one or more discrete events
    // further: this contract understands that its schema has two
discrete objects
```

```
    // that are meant to be used to send events: common, and custom
    // ledger has to have common section
    stateOut := deepMerge(map[string]interface{}(argsMap),
                          map[string]interface{}(ledgerMap))
    log.Debugf("updateAsset assetID %s merged state: %s", assetID,
stateOut)

    // handle compliance section
    alerts := newAlertStatus()
    a, found := stateOut["alerts"] // is there an existing alert
state?
    if found {
        // convert to an AlertStatus, which does not work by type
assertion
        log.Debugf("updateAsset Found existing alerts state: %s", a)
        // complex types are all untyped interfaces, so require con-
version to
        // the structure that is used, but not in the other direc-
tion as the
        // type is properly specified
        alerts.alertStatusFromMap(a.(map[string]interface{}))
    }
    // important: rules need access to the entire calculated state
    if ledgerMap.executeRules(&alerts) {
        // true means noncompliant
        log.Noticef("updateAsset assetID %s is noncompliant", as-
setID)
        // update ledger with new state, if all clear then delete
        stateOut["alerts"] = alerts
        delete(stateOut, "incompliance")
    } else {
        if alerts.AllClear() {
            // all false, no need to appear
            delete(stateOut, "alerts")
        } else {
            stateOut["alerts"] = alerts
        }
        stateOut["incompliance"] = true
    }

    // save the original event
    stateOut["lastEvent"] = make(map[string]interface{})
    stateOut["lastEvent"].(map[string]interface{})["function"] =
"updateAsset"
    stateOut["lastEvent"].(map[string]interface{})["args"] = args[0]

    // Write the new state to the ledger
    stateJSON, err := json.Marshal(ledgerMap)
    if err != nil {
        err = fmt.Errorf("updateAsset AssetID %s marshal failed:
%s", assetID, err)
        log.Error(err)
        return nil, err
    }

    // finally, put the new state
    err = stub.PutState(assetID, []byte(stateJSON))
    if err != nil {
        err = fmt.Errorf("updateAsset AssetID %s PUTSTATE failed:
%s", assetID, err)
        log.Error(err)
        return nil, err
    }
    err = pushRecentState(stub, string(stateJSON))
    if err != nil {
        err = fmt.Errorf("updateAsset AssetID %s push to re-
centstates failed: %s",
           assetID, err)
        log.Error(err)
        return nil, err
    }

    // add history state
    err = updateStateHistory(stub, assetID, string(stateJSON))
    if err != nil {
        err = fmt.Errorf("updateAsset AssetID %s push to history
failed: %s",
           assetID, err)
        log.Error(err)
        return nil, err
    }

    // NOTE: Contract state is not updated by updateAsset

  return nil, nil
}
```